Appendix.

```
#define NAP_CMD_LOGIN_LENGTH        128     /* bytes */
#define NAP_CMD_NEWADDR_LENGTH      64      /* (username + 17) bytes */

/* These servers are the responses to a DNS lookup of "server.napster.com" */
#define NAP_NUM_SERVERS     6
static int sNapServerTable[NAP_NUM_SERVERS] = {
                0x407c2910,     /* IP 64.124.41.16 */
                0x407c2911,     /* IP 64.124.41.17 */
                0x407c2912,     /* IP 64.124.41.18 */
                0x407c2913,     /* IP 64.124.41.19 */
                0xd0b8d8de,     /* IP 208.184.216.222 */
                0xd0b8d8df,     /* IP 208.184.216.223 */
};

/* These are commonly used Napster port numbers */
#define NAP_NUM_INIT_PORTS 1
static int sNapInitPortTable[NAP_NUM_INIT_PORTS] = { 8875 } ;
#define NAP_NUM_CMD_PORTS       1
static int sNapCmdPortTable[NAP_NUM_CMD_PORTS] = { 7777 }  ;

/* Note: the services table already has an entry for 8888 */
/* Note that Napster 2.0 Beta 5 also used ports 4444, 5555, and 6666 */
#define NAP_NUM_DATA_PORTS    2
static int sNapDataPortTable[NAP_NUM_DATA_PORTS] = { 6688, 6700 } ;

/* Note: the services table already has an entry for 6699 */
napFirstLook() {

    /* Temp flags */
    BOOL   probablyNapInit = FALSE ;
    char*dataStart ;        /* The start of the payload data unit */
    UINT16 cmdLength ;
    UINT16 cmdType ;
    struct napFlowListData curFlow ;
    struct napFlowListData *savedFlow ;
    NessFlowPtr nf ;
    SIDE        side1, side2 ;
    int         port2 ;
    int         addr2 ;
    int         i ;

    /* If this is a TCP flow then check to see if it is this Application */
    if (ipf->tcb) {

        /* Set shortcuts */
        dataStart = ipf->header.dataStart ;
        nf = nessFlowFromIPF(ipf) ;
        side1 = bcb_direction(ipf->bcb);
        side2 = (side1 ? 0 : 1);
        port2 = nf->conn.port[side2];
        addr2 = nf->conn.addr[side2];

/* --------- Look for connections to server.napster.com ----------- */

        if ( rsp == 1 ) {
```

```
                /* Check known port numbers */
                for (i=0; i<NAP_NUM_INIT_PORTS; i++) {
                    if (port2 == sNapInitPortTable[i]) {
                        probablyNapInit = TRUE ;
                        break ;
                    }
                }

                /* Check known addresses */
                if (!probablyNapInit) {
                    for (i=0; i<NAP_NUM_SERVERS; i++) {
                        if (addr2 == sNapServerTable[i]) {
                            probablyNapInit = TRUE ;
                            break ;
                        }
                    }
                }

                if (probablyNapInit) {
                    debugBlurt("Detected Napster Server");

                    /* Try to dig out the redirection */
                    napParseInitialConnection( ipf->header.dataLen, dataStart) ;
                    return SVC_NAP_INIT ;
                }
            }

            /* Inspect the first 5 packets for Napster, then give up */
            if ( (req + rsp) <= 5 ) {

/* ------------ Look for Napster Login ---------------------- */

                /* This is a safety net in case we missed the Init traffic */
                /* Anything on these ports will be classified as Napster-Cmd
                 * even though they are not registered to Napster. */

                for (i=0; i<NAP_NUM_CMD_PORTS; i++) {
                    if (port2 == sNapCmdPortTable[i]) {
                        return SVC_NAP_CMD ;
                    }
                }

                /* Otherwise inspect the flow for things that look like
                    Napster commands */
                if (ipf->header.dataLen >= 4) {

                    /* If the first two bytes contain the length of the packet
                        minus 4 bytes, then check for a login 'type' value */
                    /* Note: The fields are not in network byte order, they are
                        in little-endian form. */

                    cmdLength = (UINT16)((dataStart[1] << 8) + dataStart[0]) ;

                    if (cmdLength == (ipf->header.dataLen - 4) ) {

                        /* Check for a known login 'type' */
```

```
                cmdType = (UINT16)((dataStart[3] << 8) + dataStart[2]) ;

                /* Currently there are two login messages, types 2 and 6 */
                if (((cmdType == 2) || (cmdType == 6)) &&
                    (napParseCmdLogin(ipf, cmdLength, (dataStart+4)))) {
                    debugBlurt("Detected a Napster-ish login ");
                    return SVC_NAP_CMD ;
                }
            }
        }

/* ---------- Look for raw Napster Data flows ---------- */

        /*
         * Anything on these ports will be classified as Napster-Data
         *   even though they are not registered to Napster.
         */
        for (i=0; i<NAP_NUM_DATA_PORTS; i++) {
            if (port2 == sNapDataPortTable[i]) {
                return SVC_NAP_DATA ;
            }
        }

        /* Otherwise inspect the flow for things that look like
            Napster downloads or uploads */
        /* If data length is 1 byte and that byte is the ASCII "1"
                then check the response in the next packet */
        if ((ipf->header.dataLen == 1) &&
            (*ipf->header.dataStart == '1')) {

            /* Save this flow in a list of potential Napster flows */
            if (!(savedFlow =
                    (struct napFlowListData *)kmalloc(
                            sizeof(struct napFlowListData), M_NESS))) {
                info0("Unable to save Napster-Data info");

                /* Bail out because we're out of memory */
                return SVC_UNKNOWN ;
            }
            savedFlow->connBlk   = ipf->tcb ;
            savedFlow->connSeq   = ipf->tcb->connSeq ;
            nessListAdd( sFlowList, savedFlow ) ;
            return MORE_MAGIC ;
        }

        /* If the response contains the single word "GET" or "SEND"
            then this is probably Napster data (upload/download) */
        if ( ((ipf->header.dataLen == 3) && (dataStart[0] == 'G') &&
                                            (dataStart[1] == 'E') &&
                                            (dataStart[2] == 'T'))  ||

             ((ipf->header.dataLen == 4) && (dataStart[0] == 'S') &&
                                            (dataStart[1] == 'E') &&
                                            (dataStart[2] == 'N') &&
                                            (dataStart[3] == 'D')) )  {
```

```
                curFlow.connBlk = ipf->tcb ;
                curFlow.connSeq = ipf->tcb->connSeq ;
                savedFlow = (struct napFlowListData *)
                                    (nessListLookup( sFlowList, &curFlow )) ;

                /* If this flow is in the list of suspected Napster's
                    then we probably have Napster Data traffic. */
                if (savedFlow) {
                    return SVC_NAP_DATA ;
                }
            }

/* ------------------------------------------------------------*/

        }
    }
    /* If we make it to here then we did not recognize the traffic */
    return SVC_UNKNOWN ;
}


/*
 * Parse the Napster Command flow for download requests.
 */
napParseFlow ( ) {

    char   *dataStart ;      /* The start of the payload data unit */
    UINT16 cmdLength ;
    UINT16 cmdType ;

    whereStr("napParseFlow");
    debugBlurt("Parsing Napster-Cmd flow");

    if (!gNapCmdParsingEnabled) {
        return FALSE ;
    }
    debugAssert(ipf->ucb || ipf->tcb);

    if (ipf->tcb) {

        /* Look for Napster commands */
        if (ipf->header.dataLen >= 4) {
            dataStart = ipf->header.dataStart ;

            /* If the first two bytes contain the length of the packet
                minus 4 bytes, then check for a resonable type value */
            /* Note: The fields are not in network byte order, they are
                in little-endian form, so we can't use ntoh(). */

            cmdLength = (UINT16) ((((UINT16)dataStart[1] << 8) & 0xff00) |
                                        ((UINT16)dataStart[0] & 0x00ff)) ;

            /* Check for a known 'type' */
            cmdType = (UINT16) ((((UINT16)dataStart[3] << 8) & 0xff00) |
                                        ((UINT16)dataStart[2] & 0x00ff)) ;
```

```
            if (cmdLength == (ipf->header.dataLen - 4) ) {

                /* Currently the known command types are from 0 to 910,
                    but I'm allowing some room to grow */
                if (cmdType < 5000) {
                    napParseCmd( ipf, cmdLength, cmdType, (dataStart + 4)) ;
                }
            }
        }
    }
    /* Peek at this flow for as long as it exists. */
    return TRUE ;
}


/*
 * The first thing the Napster client does when it starts is make a connection
 * to the napster.com site where it is redierected to a server farm.
 */
BOOL
napParseInitialConnection( UINT16 len, char *dataStart ) {

    /* The initial connection just contains an IP address and a port number.
        ie. 208.184.216.187:7777 terminated by an ascii newline (0x0A) and
        a null terminator (0x00) (maximum of 23 chars) */
    IP_ADDR    serverAddr ;
    char*portString ;
    UINT16 portNumber ;
    BOOL   ret ;
    int  i ;

    whereStr("napParseInitialConnection");

    portString    = dataStart ;
    serverAddr  = 0 ;
    portNumber  = 0 ;
    ret         = FALSE ;

    /* If it's too long then something is wrong, bail out */
    if (len > 25) {
        return FALSE ;
    }

    for (i=0;i<len;i++) {
        if (dataStart[i] == ':') {
            dataStart[i] = '\0' ;
            serverAddr = inet_addr_in_host_order( dataStart ) ;
            dataStart[i] = ':' ;
            portString = &(dataStart[i+1]) ;
        }
        if (dataStart[i] == '\n') {
            dataStart[i] = '\0' ;
            ret = napParsePort( portString, &portNumber ) ;
            dataStart[i] = '\n' ;
            break ;
```

```
        }
    }

    if (serverAddr && ret && portNumber) {

        /* Remember this future flow */
        if (nessRememberRedirection(serverAddr, portNumber, TCP_PTCL,
                SVC_NAP_CMD)) {
            debugBlurt3("Remembering: 0x%x, %d, %d is Napster", serverAddr, portNumber,
                    SVC_NAP_CMD);
        }
        else {
            debugBlurt("DEBUG: failed to remember redirection");
        }
        return TRUE ;
    }
    return FALSE ;
}


/*
 * Once the client establishes a connection to the database server, it
 * interacts via a mesage typed protocol.  Each message begins with a
 * 2 byte length, then a 2 byte type, then the data.  This function
 * directs the data parsing based on the message type code.
 * Note: there can be more than one message in the TCP packet and the
 * messages can cross packet boundaries, but this code only looks at
 * the first message, and split messages are not processed.
 */
BOOL
napParseCmd( IPF_INFO_PTR ipf, UINT16 cmdLength, UINT16 cmdType, char *cmdDataStart)
{
    BOOL ret ;

    whereStr("napParseCmd");

    switch (cmdType) {
        case 2:
        case 6:
            debugBlurt1("Napster Login command: %d", cmdType );
            ret = napParseCmdLogin( ipf, cmdLength, cmdDataStart ) ;
            break ;
        case 216:
            debugBlurt1("Napster Search command %d", cmdType );
            ret = napParseCmdNewAddr( cmdLength, cmdDataStart ) ;
            break ;
        case 204:
        case 501:
            debugBlurt1("Napster Download/Upload command %d", cmdType );
            ret = napParseCmdNewAddr( cmdLength, cmdDataStart ) ;
            break ;
        case 300:
        case 703:
            debugBlurt1("Napster Set Port command %d", cmdType );
            ret = napParseCmdNewPort( ipf, cmdLength, cmdDataStart ) ;
            break ;
```

```
            case 901:
                debugBlurt1("Napster Listen Test command %d", cmdType );
                ret = napParseCmdNewPort( ipf, cmdLength, cmdDataStart ) ;
                break ;
            default:
                ret = FALSE ;
        }

        return ret ;
}


/*
 * The client login in message format is:
 * <username> <password> <port> "<client-info>" <link-type>
 */
BOOL
napParseCmdLogin( IPF_INFO_PTR ipf, UINT16 cmdLength, char *cmdDataStart ) {

        char copyString[NAP_CMD_LOGIN_LENGTH];
        char        *copyStringKmem ;
        char        *marker ;
        char        *userString ;
        char        *passString ;
        char        *portString ;
        UINT16      portNumber ;
        NessFlowPtr nf ;
        SIDE sside, dside;
        unsigned long saddr;

        whereStr("napParseCmdLogin");

        copyStringKmem = NULL;
        if (cmdLength < NAP_CMD_LOGIN_LENGTH) {
            memcpy(copyString, cmdDataStart, cmdLength) ;

            /* Must be null terminated for strtok_r */
            copyString[cmdLength] = 0;

            userString = strtok_r( copyString, " ", &marker);
            passString = strtok_r( NULL, " ", &marker);
            portString = strtok_r( NULL, " ", &marker);
        } else {

            /* cmdLength will be less than a packet in size */
            if (!(copyStringKmem = (char *)kmalloc( cmdLength + 1, M_NESS))) {

                /* Bail out because we're out of memory  */
                return FALSE ;
            }
            memcpy( copyStringKmem, cmdDataStart, cmdLength ) ;

            /* Must be null terminated for strtok_r */
            copyStringKmem[cmdLength] = 0 ;
            userString = strtok_r( copyStringKmem, " ", &marker) ;
            passString = strtok_r( NULL, " ", &marker) ;
```

```
            portString = strtok_r( NULL, " ", &marker) ;
        }

        /* Convert the port string into an integer */
        if ( portString && napParsePort( portString, &portNumber )) {

            /* Now we can set some associative rememberance */
            /* If the port number is zero (which means that this client is
                behind a firewall) then there is no need to remember it  */
            if (portNumber) {
                nf = nessFlowFromIPF(ipf) ;
                sside = bcb_direction(ipf->bcb) ;
                dside = (sside ? 0 : 1) ;
                saddr = nf->conn.addr[dside] ;

                /* Remember this pending flow */
                if (nessRememberDependantRedirection(saddr, portNumber, TCP_PTCL,
                        SVC_NAP_DATA, nf)) {
                    debugBlurt3("Remembering Dependant Flow: 0x%x, %d, %d is Napster-Data",
                            saddr, portNumber, TCP_PTCL);
                }
                else {
                    debugBlurt("DEBUG: failed to remember redirection");
                }
            }
            if (copyStringKmem) kfree( copyStringKmem ) ;
            return TRUE ;
        }
        if (copyStringKmem) kfree( copyStringKmem ) ;
        return FALSE ;
    }


    /*
     * The download messages are of the form:
     *  <username> <ip> <port> ...
     */
    BOOL
    napParseCmdNewAddr( UINT16 cmdLength, char *cmdDataStart ) {

        charcopyString[NAP_CMD_NEWADDR_LENGTH];
        char        *copyStringKmem ;
        char        *marker ;
        char        *userString ;
        char        *addrString ;
        char        *portString ;
        UINT16      portNumber ;
        IP_ADDR     peerAddr ;

        whereStr("napParseCmdNewAddr");

        copyStringKmem = NULL;

        /*
         * Look at the first NAP_CMD_NEWADDR_LENGTH bytes of the command.
         * Considering that we're only interested in the first three fields
```

```
 * of the command and that the <ip> and <port> pieces will only consume
 * a maximum of 17 bytes, including white space, then we only need
 * to account for a big username length.  If it looks like the
 * username is so big that we may truncate the ip addr or port number,
 * then we must use kmalloc to correctly parse the command.
 */

/* cmdLength will be less than a packet in size */
memcpy(copyString, cmdDataStart, NAP_CMD_NEWADDR_LENGTH ) ;

/* Must be null terminated for strtok_r */
copyString[NAP_CMD_NEWADDR_LENGTH-1] = 0 ;
userString = strtok_r( copyString, " ", &marker) ;

/* Check to see if the username was a resonable length */
if ((marker - copyString) < (NAP_CMD_NEWADDR_LENGTH - 17)) {

    /* If username is okay, then get the remaining pieces */
    addrString = strtok_r( NULL, " ", &marker) ;
    portString = strtok_r( NULL, " ", &marker) ;
}

/* Otherwise, it is an obnoxiously long username and we must use kmalloc */
else {

    /* cmdLength will be less than a packet in size */
    if (!(copyStringKmem = (char *)kmalloc( cmdLength + 1, M_NESS))) {

        /* Bail out because we're out of memory */
        return FALSE ;
    }
    memcpy( copyStringKmem, cmdDataStart, cmdLength ) ;

    /* Must be null terminated for strtok_r */
    copyStringKmem[cmdLength] = 0 ;
    marker = NULL ;
    userString = strtok_r( copyStringKmem, " ", &marker) ;
    addrString = strtok_r( NULL, " ", &marker) ;
    portString = strtok_r( NULL, " ", &marker) ;
}

/* Convert the port string into an integer */
if ( addrString && portString && napParsePort(portString, &portNumber) ) {

    /* Convert the address string to an IP_ADDR */
    peerAddr = (UINT32)strtoul(addrString, (char **)0, 10) ;

    if (peerAddr != ULONG_MAX) {
        peerAddr = rev4(peerAddr) ;

        /* Since this is a specific download command that will happen
            right now, we can use the regular rememberance */
        if (peerAddr && portNumber) {

            /* Remember this future flow */
            if (nessRememberRedirection(peerAddr, portNumber, TCP_PTCL,
```

```
                    SVC_NAP_DATA)) {
                debugBlurt3("Remembering: 0x%x, %d, %d is Napster-Data", peerAddr,
                        portNumber, SVC_NAP_DATA);
            }
            else {
                debugBlurt("DEBUG: failed to remember redirection");
            }
        }
    }
        if (copyStringKmem) kfree( copyStringKmem ) ;
    return TRUE ;
}
if (copyStringKmem) kfree( copyStringKmem ) ;
return FALSE ;
}


/*
 * Some messages change or add port numbers and just contain the number:
 * <port>
 */
BOOL
napParseCmdNewPort( IPF_INFO_PTR ipf, UINT16 cmdLength, char *cmdDataStart ) {

    UINT16      portNumber ;
    NessFlowPtr nf ;
    SIDE sside, dside;
    unsigned long saddr;

    whereStr("napParseCmdNewPort");

    /* Convert the port string into an integer */
    if ( napParsePort( cmdDataStart, &portNumber ) && portNumber) {

        /* Now we can set some associative rememberance */
        nf = nessFlowFromIPF(ipf) ;
        sside = bcb_direction(ipf->bcb) ;
        dside = (sside ? 0 : 1) ;
        saddr = nf->conn.addr[dside] ;

        debugBlurt2("PENDING FLOW : client 0x%x, port %d",
                    saddr, portNumber );

        /* Remember this pending flow */
        if (nessRememberDependantRedirection(saddr, portNumber, TCP_PTCL,
                SVC_NAP_DATA, nf)) {

    debugBlurt3("Remembering Dependant Flow: 0x%x, %d, %d is Napster-Data",
                    saddr, portNumber, TCP_PTCL);
        } else {
            debugBlurt("DEBUG: failed to remember redirection");
        }
        return TRUE ;
    }
    return FALSE ;
}
```

United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division

Application deficiencies found during scanning:

☐ Page(s)_____ of _____ were not present for scanning. (Document title)

☐ Page(s)_____ of _____ were not present for scanning. (Document title)

Specification contains appendix in pages 15-25.

☐ ***Scanned copy is best available.***